

Clase 5 — 02.12.25

#JAVA



Profesor: José Antonio Martín



Unidad: Programación Multiproceso



Fecha: 02/12/2025



Tema: Ejemplos de programación de procesos en Java | | Inicio del tema Programación Multihilo

1 Contexto de la clase

Esta clase sirve como **cierre práctico** de todo el bloque de **programación multiproceso**, aplicando los conceptos vistos en Linux y sistemas operativos al **lenguaje Java**.

La idea clave es demostrar que:

- Un proceso no es algo abstracto del sistema operativo.
- Desde Java podemos **crear, gestionar, monitorizar y finalizar procesos reales del sistema**.
- Todo lo visto anteriormente (PID, estado, señales, foreground/background) **tiene un reflejo directo en código**.

A partir de aquí, el profesor enlaza de forma natural con el siguiente tema: **programación multihilo**.

2 Ejemplo 1 — Creación de un proceso con `Runtime.exec()`

Este ejemplo muestra la forma **más directa y clásica** de crear un proceso externo desde Java. Aunque hoy se recomienda `ProcessBuilder`, este método sigue siendo importante porque ayuda a entender **qué es realmente un proceso desde Java**.

Código completo del ejemplo

```
package Ejemplo1;

public class Creacion1 {

    public static void main(String[] args) throws Exception {

        // Lanzamos un proceso externo (Google Chrome)
        Process proceso1 = Runtime.getRuntime()
            .exec("C:\\Program Files\\Google\\Chrome\\Application\\chrome.exe");

        // Mostramos información básica del proceso
        System.out.println("El PID del proceso 1 es: " + proceso1.pid());
        System.out.println("El proceso 1 está activo: " + proceso1.isAlive());
        System.out.println("Información adicional del proceso 1: " + proceso1.info());

        // Esperamos 5 segundos
        Thread.sleep(5000);

        // Finalizamos el proceso
```

```
        proceso1.destroy();

        // Comprobamos si sigue activo
        System.out.println("El proceso 1 está ahora activo: " + proceso1.isAlive());
    }
}
```

Salida esperada en la terminal

La salida puede variar ligeramente, pero será similar a esta:

```
El PID del proceso 1 es: 36668
El proceso 1 está activo: true
Información adicional del proceso 1:
[user: Optional[USUARIO],
 cmd: C:\Program Files\Google\Chrome\Application\chrome.exe,
 startTime: Optional[2025-12-02T16:07:42.365Z]]
El proceso 1 está ahora activo: false
```

Explicación de las líneas clave del código

Creación del proceso

```
Process proceso1 = Runtime.getRuntime()
    .exec("C:\\Program Files\\Google\\Chrome\\Application\\chrome.exe");
```

- `Runtime.getRuntime()` devuelve la **instancia única del entorno de ejecución de Java** asociada a la JVM actual.
- `exec()` solicita al **sistema operativo** la creación de un nuevo proceso.
- El proceso creado **no pertenece a Java**, sino al sistema operativo.
- Java recibe un objeto `Process` que actúa como **descriptor** o **manejador** del proceso externo.

👉 Aquí se materializa el concepto de **proceso padre (Java)** y **proceso hijo (Chrome)**.

Obtención del PID

```
proceso1.pid();
```

- Devuelve el **PID real del sistema operativo**, no un identificador interno de Java.
- Este PID:
 - Coincide con el mostrado en el Administrador de tareas (Windows).
 - Coincide con `ps / top` en Linux.
- Permite relacionar directamente Java con la gestión de procesos a bajo nivel.

👉 Conecta directamente con la teoría de PID del kernel.

Comprobación del estado

```
proceso1.isAlive();
```

- Devuelve `true` si el proceso:
 - Ha sido creado correctamente.
 - No ha terminado aún.
- Devuelve `false` si:
 - El proceso ya ha finalizado.
 - Ha sido destruido.

Este método **no comprueba errores**, solo existencia y estado de vida.

Información avanzada del proceso

```
proceso1.info();
```

- Devuelve un objeto `ProcessHandle.Info`.
- Contiene información que **proviene del sistema operativo**, no de Java:
 - Usuario que ejecuta el proceso.
 - Comando exacto.
 - Hora de inicio.
 - Tiempo de CPU (si está disponible).

👉 Es el equivalente conceptual a inspeccionar un proceso con herramientas del sistema.

Pausa del hilo principal

```
Thread.sleep(5000);
```

- Suspende **solo el hilo principal de Java**.
- El proceso externo **sigue ejecutándose**.
- Demuestra que:
 - Proceso \neq hilo.
 - No existe dependencia directa entre ambos.

Finalización del proceso

```
proceso1.destroy();
```

- Solicita una **finalización ordenada**.
- Es similar a:
 - `SIGTERM` en Linux.
- El proceso puede:
 - Aceptar el cierre.
 - Tardar unos milisegundos en cerrarse.

No garantiza una finalización inmediata.

Aspectos importantes a destacar

- `Runtime.exec()` crea procesos, **no hilos**.
- El proceso lanzado no depende del hilo principal.
- `Thread.sleep()` solo bloquea el hilo de Java, **no el proceso externo**.
- `destroy()` solicita un cierre ordenado, no forzado.

3 Ejemplo 2 — Uso de `ProcessBuilder` y finalización forzada

Este ejemplo introduce `ProcessBuilder`, que es la **forma recomendada** de trabajar con procesos en Java moderno. Permite mayor control y encaja mejor con sistemas reales.

Código completo del ejemplo

```
package Ejemplo2;

public class Creacion2 {

    public static void main(String[] args) throws Exception {

        // Creamos el proceso Chrome con ProcessBuilder
        ProcessBuilder constructor1 =
            new ProcessBuilder("C:\\Program
Files\\Google\\Chrome\\Application\\chrome.exe");
        Process proceso1 = constructor1.start();

        System.out.println("El PID del proceso 1 es: " + proceso1.pid());
        System.out.println("El proceso 1 está activo: " + proceso1.isAlive());
        System.out.println("Información adicional del proceso 1: " + proceso1.info());

        // Esperamos 5 segundos
        Thread.sleep(5000);

        // Creamos otro proceso que mata Chrome
        ProcessBuilder constructor2 =
            new ProcessBuilder("taskkill", "/F", "/IM", "chrome.exe");
        Process eliminaChrome = constructor2.start();

        // Esperamos a que termine taskkill
        eliminaChrome.waitFor();

        System.out.println("El proceso 1 está ahora activo: " + proceso1.isAlive());
    }
}
```

Salida esperada en la terminal

```
El PID del proceso 1 es: 37124
El proceso 1 está activo: true
```

```
Información adicional del proceso 1:  
[user: Optional[USUARIO],  
  cmd: C:\Program Files\Google\Chrome\Application\chrome.exe,  
  startTime: Optional[2025-12-02T16:15:01.102Z]]  
El proceso 1 está ahora activo: false
```

(Chrome se cerrará de forma inmediata y forzada.)

Qué está pasando a nivel conceptual

Aquí ocurre algo clave a nivel de sistema operativo:

- Java **no mata directamente** el proceso Chrome.
- Java crea **otro proceso del sistema** (`taskkill`).
- Ese proceso es el que envía la orden de finalización forzada.

Es exactamente el mismo concepto que:

- `kill -9` en Linux.
- `pkill` o `killall` .

👉 Un proceso puede crear **otros procesos para gestionar procesos**.

Explicación de las líneas clave del código

Creación del proceso con `ProcessBuilder`

```
ProcessBuilder constructor1 =  
    new ProcessBuilder("C:\\Program  
Files\\Google\\Chrome\\Application\\chrome.exe");
```

- `ProcessBuilder` encapsula:
 - El comando.
 - Los argumentos.
 - El entorno.
- Es más flexible que `Runtime.exec()` .
- Permite redirecciones y control avanzado.

👉 Es la API recomendada para programación multiproceso en Java.

```
Process procesos1 = constructor1.start();
```

- `start()` solicita al sistema la creación del proceso.
- Devuelve un objeto `Process` .
- Si el comando es inválido → lanza excepción.

Creación de un segundo proceso para matar el primero

```
ProcessBuilder constructor2 =  
    new ProcessBuilder("taskkill", "/F", "/IM", "chrome.exe");
```

- Aquí Java **no mata Chrome directamente**.
- Crea un proceso auxiliar:
 - `taskkill` es un programa del sistema.
 - `/F` fuerza la finalización.
 - `/IM` indica el nombre del proceso.

👉 Es el mismo concepto que `killall` o `pkill` en Linux.

```
Process eliminaChrome = constructor2.start();
```

- Se crea **otro proceso independiente**.
- Java ahora controla **dos procesos distintos**:
 - Chrome.
 - `taskkill`.

Esto demuestra que un proceso puede gestionar otros procesos.

Sincronización con `waitFor()`

```
eliminaChrome.waitFor();
```

- Bloquea el hilo principal hasta que:
 - `taskkill` haya terminado.
- Garantiza que Chrome ha sido cerrado antes de continuar.
- Evita condiciones de carrera.

👉 Es sincronización entre procesos, no entre hilos.

Diferencias clave respecto al ejemplo 1

- Se usa `ProcessBuilder`, no `Runtime`.
- La finalización es **forzada** (`/F`).
- Se delega explícitamente la acción al sistema operativo.
- Se espera a que el proceso de eliminación termine (`waitFor()`).

Este ejemplo conecta directamente con **señales, permisos y gestión avanzada de procesos**.

4 Ejemplo 3 — Comunicación entre procesos (entrada y salida)

Este ejemplo introduce un concepto nuevo y muy importante: **leer la salida de un proceso externo** desde Java.

El profesor comenta correctamente que **no se ejecuta en su entorno** porque la ruta o el classpath no son correctos.

Código completo del ejemplo

```

package Ejemplo3;

import java.io.*;

public class Creacion3 {

    public static void main(String[] args) throws Exception {

        // Intentamos ejecutar otro programa Java como proceso externo
        ProcessBuilder constructor1 =
            new ProcessBuilder("java", "Ejemplo3.HolaMundo");

        Process proceso1 = constructor1.start();

        // Capturamos la salida estándar del proceso
        InputStream entradaProceso1 = proceso1.getInputStream();
        BufferedReader reader =
            new BufferedReader(new InputStreamReader(entradaProceso1));

        String linea;
        while ((linea = reader.readLine()) != null) {
            System.out.println(linea);
        }

        // Esperamos a que el proceso termine
        proceso1.waitFor();

        System.out.println("El proceso \"Hola Mundo\" está activo: " +
proceso1.isAlive());
    }
}

```

Por qué no se ejecuta correctamente en clase

Para que este código funcione correctamente, es necesario que:

- El archivo `HolaMundo.class` exista.
- Esté en el classpath correcto.
- La estructura de paquetes sea accesible.
- El comando `java Ejemplo3.HolaMundo` funcione desde consola.

Si alguno de estos puntos falla, el proceso:

- Se lanza.
- Falla internamente.
- No produce salida útil.

Salida esperada si se ejecutara correctamente

Suponiendo que `HolaMundo` imprime un mensaje:

```

System.out.println("Hola Mundo desde otro proceso");

```

La salida sería:

```
Hola Mundo desde otro proceso  
El proceso "Hola Mundo" está activo: false
```

Explicación de las líneas clave del código

Lanzar un programa Java como proceso externo

```
new ProcessBuilder("java", "Ejemplo3.HolaMundo");
```

- Se invoca el **intérprete de Java** como programa externo.
- `Ejemplo3.HolaMundo` es la clase a ejecutar.
- Requiere:
 - Clase compilada.
 - Classpath correcto.
 - Ruta válida.

Si falla alguno → el proceso se crea, pero termina con error.

Acceso a la salida estándar del proceso

```
proceso1.getInputStream();
```

- Devuelve la **salida estándar (stdout)** del proceso hijo.
- Es lo que el proceso imprimiría por consola.
- Desde Java, se accede como un `InputStream`.

👉 El nombre es confuso:

`InputStream` porque **Java lee**, aunque el proceso escriba.

Lectura estructurada de la salida

```
BufferedReader reader =  
    new BufferedReader(new InputStreamReader(entradaProceso1));
```

- `InputStreamReader` convierte bytes → caracteres.
- `BufferedReader` permite:
 - Leer línea a línea.
 - Mayor eficiencia.
- Es el patrón estándar para leer salida de procesos.

Lectura continua

```
while ((linea = reader.readLine()) != null) {  
    System.out.println(linea);  
}
```

```
}
```

- El bucle:
 - Lee hasta que el proceso termine.
 - Captura todo lo que el proceso imprima.
 - Simula el comportamiento de un **pipe** en Linux.
-

Espera a la finalización

```
proceso1.waitFor();
```

- Bloquea hasta que el proceso hijo termine.
 - Permite asegurar que:
 - Toda la salida ha sido leída.
 - El proceso ya no existe.
-

Qué demuestra este ejemplo

Este ejemplo es clave porque muestra que:

- Un proceso puede **producir datos**.
- Otro proceso puede **leer esos datos**.
- Existe comunicación entre procesos independientes.
- No se comparte memoria.
- Todo se hace mediante flujos (streams).

Esto conecta directamente con:

- Redirecciones de entrada/salida en Linux.
 - Pipes (|).
 - Comunicación entre procesos.
 - Fundamentos previos a la programación multihilo.
-

Idea clave de los tres ejemplos (resumen del profesor)

Todos los ejemplos tienen en común que:

- Trabajan con procesos reales del sistema.
 - Muestran el ciclo de vida de un proceso.
 - Permiten observar PID, estado y finalización.
 - Refuerzan la teoría de multiproceso.
 - Preparan el terreno para entender **multihilo**, donde:
 - No hay procesos separados.
 - Se comparte memoria.
 - La complejidad aumenta.
-

Resumen conceptual de los métodos usados en los ejemplos:

- `pid()`
Devuelve el identificador del proceso en el sistema operativo.
 - `isAlive()`
Indica si el proceso sigue ejecutándose.
 - `destroy()`
Solicita la finalización del proceso.
 - `destroyForcibly()`
Fuerza la finalización inmediata del proceso.
 - `waitFor()`
Bloquea el hilo hasta que el proceso termina.
 - `getInputStream()`
Permite leer la salida estándar del proceso.
 - `getErrorStream()`
Permite leer la salida de errores del proceso.
 - `info()`
Devuelve información adicional sobre el proceso.
-

6 Teoría — Monohilo vs Multihilo (introducción)

Programa monohilo

- Tiene **un único hilo de ejecución**.
- Las instrucciones se ejecutan de forma secuencial.
- Si una tarea se bloquea, todo el programa se detiene.

Ejemplo: un programa Java básico con `main`.

Programa multihilo

- Tiene **varios hilos dentro del mismo proceso**.
- Los hilos comparten memoria.
- Se pueden ejecutar tareas en paralelo.
- Requiere sincronización para evitar errores.

👉 Diferencia clave:

- **Multiproceso** → varios procesos independientes.
 - **Multihilo** → varios hilos dentro del mismo proceso.
-

7 Qué tenían en común todos los ejemplos (resumen del profesor)

Todos los ejemplos de la clase comparten:

- Creación de procesos reales del sistema.
- Obtención del PID.
- Consulta del estado del proceso.
- Esperas controladas (`sleep` , `waitFor`).

- Finalización del proceso.
- Independencia entre proceso padre e hijo.
- Aplicación directa de la teoría de procesos.

Este bloque sirve como **síntesis práctica** de toda la unidad de **Programación Multiproceso**.

8 Transición al siguiente tema

A partir de aquí, el enfoque cambia:

- Ya no se crean procesos externos.
- Se trabaja con **hilos dentro del mismo proceso Java**.
- Se comparte memoria.
- Aparecen problemas de sincronización.

👉 Comienza el tema de **Programación Multihilo**.

Conclusión y resumen de la Programación Multiproceso

A lo largo de esta unidad se ha construido una visión completa y coherente de qué es realmente la **programación multiproceso**, tanto desde el punto de vista del **sistema operativo** como desde el **lenguaje Java**.

La teoría ha mostrado que un **proceso** es la unidad básica de ejecución gestionada por el kernel: un programa en ejecución con su propio espacio de memoria, su contexto de ejecución y su ciclo de vida independiente. Cada proceso es identificado mediante un **PID**, puede encontrarse en distintos estados y puede ser planificado, detenido o finalizado por el sistema operativo.

Desde Linux se ha visto cómo:

- Los procesos se organizan jerárquicamente (padre–hijo).
- El sistema puede ejecutar múltiples procesos de forma concurrente.
- Existen herramientas para monitorizarlos (`ps` , `top` , `htop`).
- Los procesos pueden ejecutarse en **primer plano** o **segundo plano**.
- Se controlan mediante **señales** (`kill` , `pkill` , `killall`).
- Muchos procesos se ejecutan como **servicios**, gestionados por `systemd` .

Toda esta base teórica se traslada directamente a la práctica con los ejemplos en Java.

Los ejemplos de código demuestran que Java:

- No ejecuta programas externos como hilos.
- Interactúa directamente con el sistema operativo.
- Puede crear procesos reales, independientes de la JVM.
- Puede consultar su PID, estado y tiempo de ejecución.
- Puede sincronizarse con ellos (`waitFor()`).
- Puede comunicarse con ellos mediante flujos de entrada y salida.

De esta forma, el código Java actúa como un **proceso controlador**, reproduciendo exactamente el modelo visto en Linux:

- Un proceso padre crea procesos hijos.

- Los procesos se ejecutan de forma independiente.
- La comunicación se realiza mediante mecanismos externos, no memoria compartida.

Esto es especialmente importante porque marca la **frontera conceptual** entre multiproceso y multihilo:

- En **multiproceso**:
 - Cada proceso tiene su propia memoria.
 - El aislamiento es alto.
 - La comunicación es más costosa.
 - La estabilidad del sistema es mayor.
- En **multihilo**:
 - Los hilos comparten memoria.
 - La comunicación es más rápida.
 - La complejidad aumenta.
 - Aparecen problemas de sincronización y concurrencia.

Los ejemplos finales consolidan toda la teoría previa:

- PID → `process.pid()`
- Estados → `isAlive()`
- Finalización → `destroy()`, `taskkill`
- Foreground / background → procesos independientes
- Comunicación → `InputStream`, `BufferedReader`

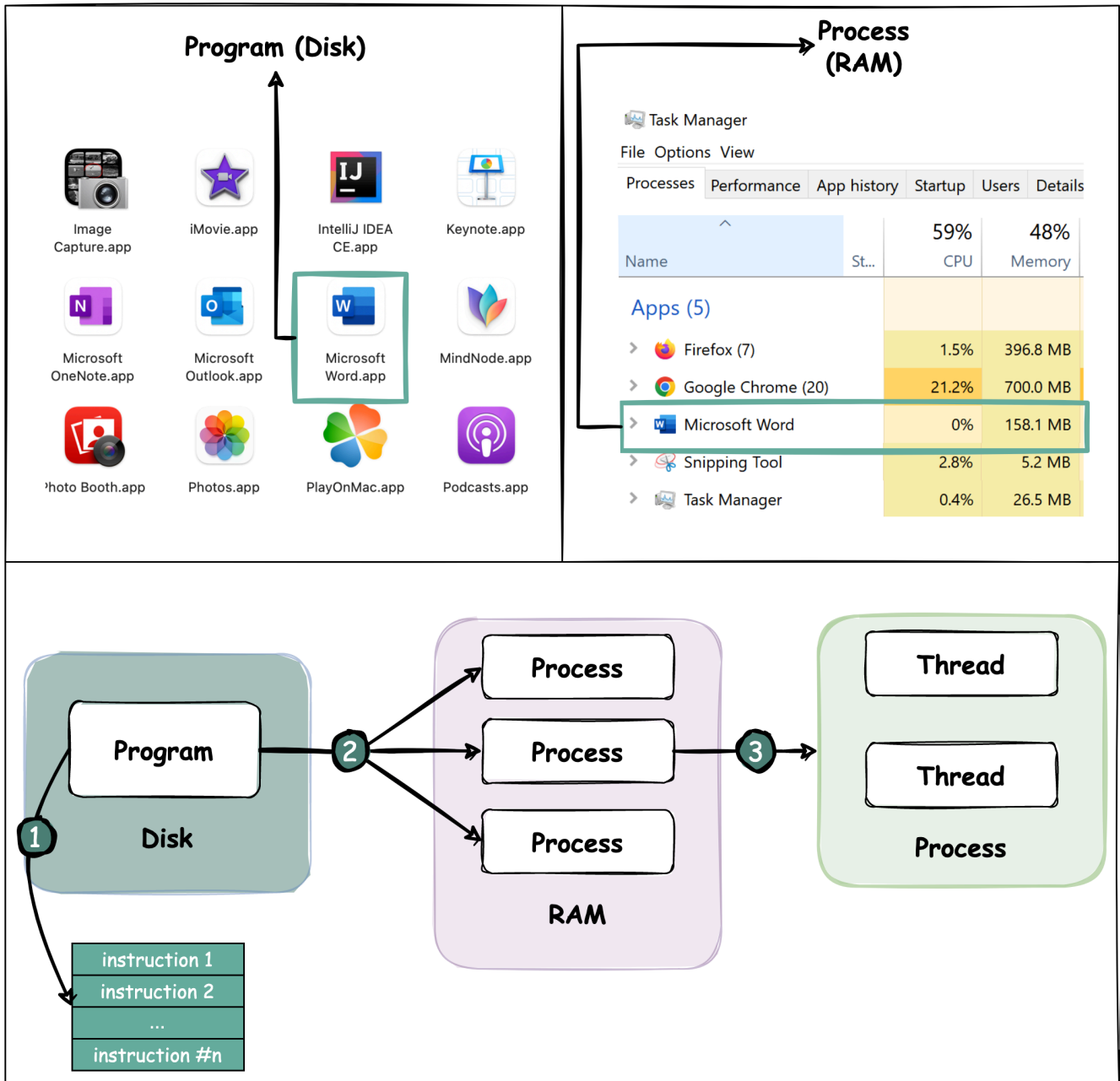
En conjunto, esta unidad demuestra que la programación multiproceso no es un concepto abstracto, sino una **realidad tangible** que se puede observar, controlar y manipular tanto desde el sistema operativo como desde un lenguaje de alto nivel como Java.

Esta base es imprescindible para comprender correctamente el siguiente bloque del curso: **la programación multihilo**, donde se abandona el aislamiento entre procesos y se entra en un modelo más potente, pero también más complejo y delicado.

1 OBJETIVOS DE LA UNIDAD — PROGRAMACIÓN MULTHILO

La **programación multihilo** surge como evolución natural del modelo multiproceso, con el objetivo de mejorar el rendimiento y la capacidad de respuesta de las aplicaciones modernas.

Program vs Process vs Thread



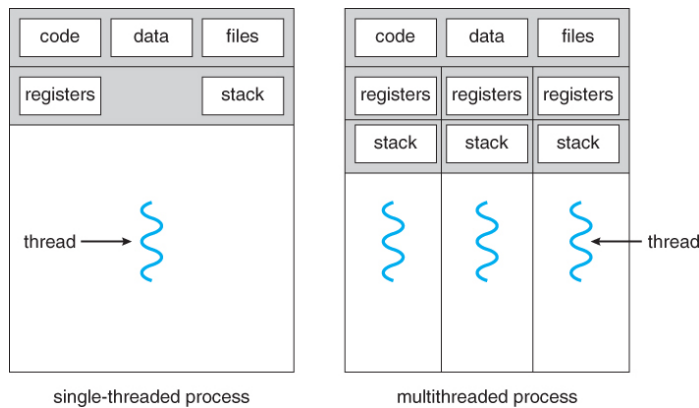
Los objetivos fundamentales de esta unidad son:

- **Mejorar el rendimiento y la utilización de recursos**, dividiendo el trabajo de una aplicación en varios hilos que pueden ejecutarse de forma concurrente.
- **Aumentar la capacidad de respuesta**, permitiendo que una aplicación siga atendiendo al usuario mientras realiza tareas costosas en segundo plano.
- **Facilitar la gestión de tareas concurrentes**, estructurando una aplicación en flujos de ejecución independientes.
- **Optimizar operaciones de entrada/salida (I/O)**, aprovechando los tiempos de espera para ejecutar otros hilos.

Estos objetivos solo se comprenden correctamente al analizar cómo funcionan **procesos y hilos a nivel interno**, tal y como muestran las imágenes del tema.

2 INTRODUCCIÓN — CONCEPTO DE HILO

Un **hilo de ejecución** (thread, hebra o proceso ligero) es una **secuencia de instrucciones** que puede ejecutarse de forma concurrente con otras dentro de un mismo proceso.



En los sistemas operativos antiguos:

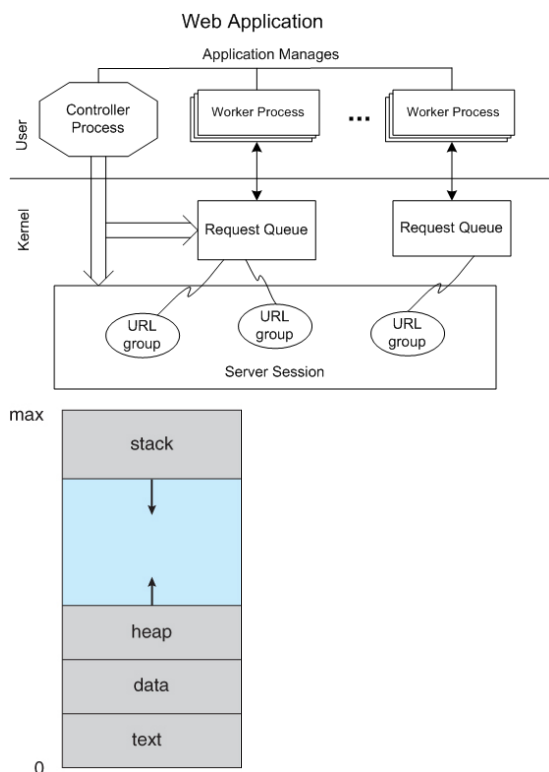
- Un proceso equivalía a un único hilo.
- No era posible realizar varias tareas simultáneas dentro de una aplicación.

En los sistemas modernos:

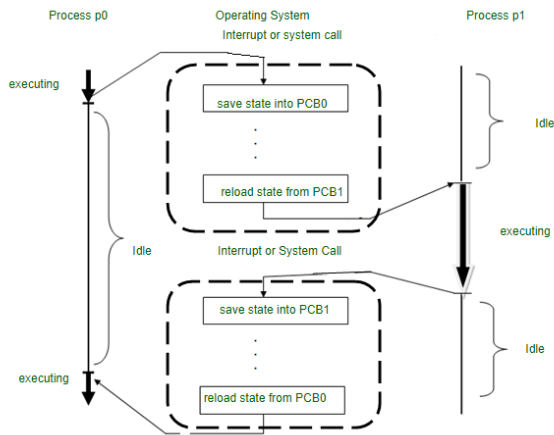
- Un proceso puede contener **múltiples hilos de control**.
- Esto permite dividir una aplicación en tareas más pequeñas que se ejecutan al mismo tiempo.

Los lenguajes actuales, como **Java**, incorporan soporte nativo para trabajar con hilos, ya que este modelo refleja cómo funcionan realmente los sistemas operativos.

3 VISIÓN EXTERNA — PROCESOS INDEPENDIENTES (P1 y P2)



En el esquema del profesor aparecen dos bloques separados, etiquetados como **P1** y **P2**, que representan **procesos distintos**.



Desde el punto de vista del sistema operativo:

- Cada proceso tiene:
 - Su propio espacio de memoria.
 - Sus propios recursos.
 - Su propio hilo principal (*main thread*).
- Los procesos son **independientes y aislados** entre sí.
- No existe acceso directo a la memoria de otro proceso.

La **✗** dibujada entre P1 y P2 indica que **no hay memoria compartida**.

Si un proceso necesita comunicarse con otro, debe hacerlo mediante mecanismos externos como pipes, sockets o ficheros.

Este modelo coincide exactamente con la **programación multiproceso** vista en el tema anterior.

4 VISIÓN INTERNA — HILOS DENTRO DE UN PROCESO

Dentro de cada proceso aparece un **Main Thread**, del que pueden surgir varios hilos secundarios.

Aquí se introduce el concepto clave del multihilo:

Todos los hilos de un mismo proceso comparten el mismo espacio de memoria.

Esto permite:

- Comunicación directa entre hilos.
- Compartición de datos sin mecanismos externos.
- Mayor eficiencia y rapidez.

Pero también implica que los errores en un hilo pueden afectar al resto del proceso.

5 RECURSOS COMPARTIDOS Y RECURSOS PROPIOS DE UN HILO

Aunque los hilos comparten muchos recursos, **no son idénticos**.

♦ Cada hilo posee:

- Su propio contador de programa.
- Su propio estado de ejecución.
- Su propia pila (*stack*).

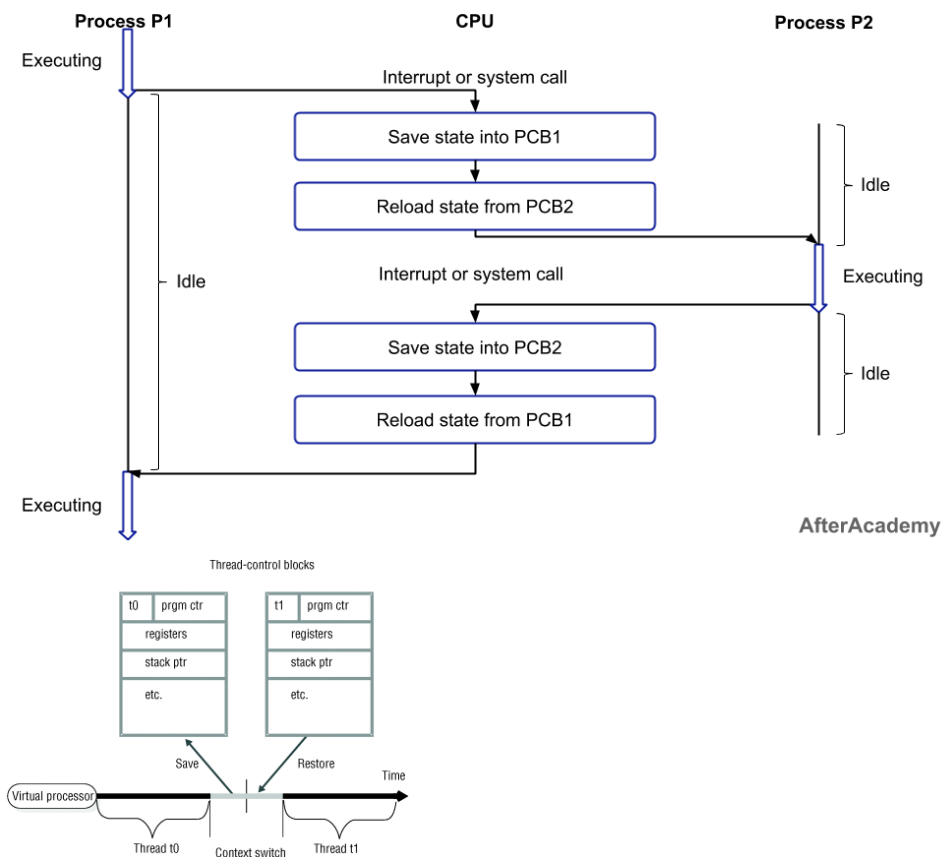
- Sus propios registros de CPU.

♦ Todos los hilos comparten:

- El código del programa.
- El heap (memoria dinámica).
- Variables globales y estáticas.
- Recursos del proceso (ficheros, sockets, etc.).

Esta separación explica por qué los hilos pueden ejecutarse de forma independiente, pero interactuar directamente con los mismos datos.

6 CAMBIO DE CONTEXTO — PROCESOS VS HILOS



El sistema operativo debe cambiar de una tarea a otra constantemente. Este cambio se denomina **cambio de contexto**.

♦ Cambio de proceso

Cuando el sistema pasa de un proceso a otro:

- Cambia el espacio de direcciones.
- Cambia la memoria.
- Cambian los recursos.
- El coste en tiempo es elevado.

➔ Se produce un **overhead alto**.

♦ Cambio de hilo

Cuando el cambio se produce entre hilos del mismo proceso:

- La memoria se mantiene.
- El espacio de direcciones es el mismo.
- Solo cambia el estado del hilo.

→ El overhead es **mucho menor**.

Por este motivo se dice que:

- Los **hilos son “baratos”**.
- Los **procesos son “costosos”**.

7 RELACIÓN DIRECTA CON EL TEMA ANTERIOR

El esquema conecta directamente con la teoría de **programación multiproceso**:

- **Multiproceso:**
 - Procesos aislados.
 - Comunicación compleja.
 - Mayor seguridad.
- **Multihilo:**
 - Hilos dentro del mismo proceso.
 - Memoria compartida.
 - Mayor rendimiento, pero más riesgo.

Ambos modelos se complementan y se usan según el tipo de problema.

8 CONCLUSIÓN DEL BLOQUE INTRODUCTORIO

Este bloque introductorio establece la idea central del tema:

La programación multihilo permite ejecutar varias tareas concurrentes dentro de un mismo proceso, compartiendo memoria y reduciendo el coste de ejecución frente al multiproceso.

Las imágenes y el esquema del profesor proporcionan la **representación mental correcta** para entender todo lo que viene a continuación: estados de un hilo, prioridades, sincronización y problemas de concurrencia.
